Санкт-Петербургский государственный университет
Факультет прикладной математики — процессов управления
Кафедра моделирования экономических систем

Забелин Денис Витальевич

Выпускная квалификационная работа бакалавра

Обнаружение и векторизация разрывных линий на цифровых изображениях

Направление 010300
Фундаментальная информатика и информационные технологии

Заведующий кафедрой,
доктор физ.-мат. наук,
профессор — Прасолов Александр Витальевич

Научный руководитель,
кандидат физ.-мат. наук,
доцент — Ковшов Александр Михайлович

Рецензент,
кандидат техн. наук,
доцент — Гришкин Валерий Михайлович

Санкт-Петербург
2017

Содержание

Введение

Данная дипломная работа представляет собой описание и реализацию алгоритмов по по обнаружению и векторизации прерывистых линий. Проект реализован на языке программирования Java в среде разработки NetBeans IDE.

Со времен начала развития компьютерное зрения был проделан большой путь вперед. В течение своего развития многие алгоритмы и методы старались подражать человеческому мышлению. Но до сих пор в некотрых областях, таких как распознавание линий и границ объектов, предпринимаются попытки обнаружения и отслеживания.

Целью данной работы является описание и реализация инструментария для обнаружения и соединения разрывных линий, представление их в векторизованном виде в памяти компьютера для последующей обработки.

В данной работе будет описано обнаружение и отслеживание прерывистых линий, которые могут состоять из разнородных сегментов. Основное свойство, которому каждый сегмент линии должен удовлетворять — продольность. В остальном — длине, ширине, кривизне — сегменты могут быть совершенно разными.

Постановка задачи

Для более удобной работы пользователя реализован графический интерфейс. Программа состоит из нескольких компонент (Contour Texture, Curve Detection Control), в которых будут реализованы все этапы алгоритма:

1. Первоначальная обработка изображения происходит в окне обнаружения конуров — Contour Texture. В первоначальную обработку входит нахождение контуров с использованием различных методов (методы среднего или пользовательского порогов, метод Оцу), а также их фильтрация по контурным признакам.

2. Обнаружение линий и их векторизация происходит в окне отслеживания линий и соединения сегментов — Curve Detection Control. Начинается с того, что производится скелетизация контуров. Затем осуществляется отслеживание отдельных скелетов, последовательно соединяя соседние пикселы в одну последовательность.

3. Соединение получившихся сегментов, удовлетворяющих двум порогам: угол и расстояние.

Также еще одно применение соединения линий, которое будет описано в главе 3 — это замыкание отслеженных линий, получившихся в результате поиска градиента изображения и использования модифицированного метода Кэнни.

Обзор литературы

Первый из источников, в котором описан большой теоретический пласт на тему работы с кривыми на изображениями, является диссертация [1]. В этой работе описывается, как нужно идентифицировать отдельную кривую и отследить ее по всему изображению, даже если она пересекает другие линии, местами исчезает и снова появляется. Также в работе описаны алгоритмы для извлечение и математической формализации кривых. К сожалению, в работе представлены только теоретические выкладки и не предоставлено реализации описаных алгоритмов для читателя.

Также существуют работы для отдельного вида линий, которые состоят из одного вида объектов. В статье [2] описан алгоритм, который отслеживает прямые на множестве точек, разбросанных на двухмерном изображении. В работе представлены принципы работы детектора прямых, включающую в себя процедуру сокращения избыточности, а также общую оценку сложности алгоритма. Следующая статья [3] является описанием отслеживания гладких кривых во множестве точек, результаты которого в данном классе изображении очень близки к человеческим способностям к распознаванию кривых на таком же классе изображений. По сравнению с предыдущей статьей, где тип линии уже известен (прямая), в этой статье на каждом шаге направление отслеживания может изменятся произвольный образом, соединяя точки в последовательность.

Описанные в [2] и [3] алгоритмы разработаны для специфичного типа пунктирны линий. Но эти алгоритмы разработанны для специфичного типа пунктирных линий, состоящих из только из точек. Конечно, это очень близко к классу пунктирных линий, состоящих только из одинкавых объектов (точки, треугольники, квадраты, штрихи и т.д.), но применение таких алгоритмов эффективно только на специфичных изображениях.

Глава 1. Обработка изображения

1.1. Контурная бинаризация

Задача бинаризации состоит в том, чтобы разделить пикселы изображения на два класса (черные и белые). Сначала исходное изображение переводится в полутоновое, где у каждого пиксела есть значение яркости от 0 до 255. В программе существует три способа бинаризации изображения. На Рис. 1 представлен исхожное изображение, к которому будут применены разные способы бинаризации. Первый способ, результат которого представлен на Рис. 3 — метод среднего порога. Высчитывается средняя интенсивность всех пикселов изображения, которая является порогом для бинаризации. Второй способ, результат которого представлен на Рис. 2 — метод Оцу [4] [5]. В методе Оцу порог подбирается таким образом, чтобы дисперсия между классами была максимальной. И последний способ — ручное изменение порога — реализован потому, что не всегда можно получить хороший результат первыми двумя методами. На Рис. 4 и Рис. 5 показан пример ручного изменения порога. В графическрм интерфейсе программы ручное изменение порога осуществляется с помощью бегунка с автоматическим отображением результата на основной панели, что будет интуитивно понятно для пользователя.

Рис. 1: Исходное ихображение

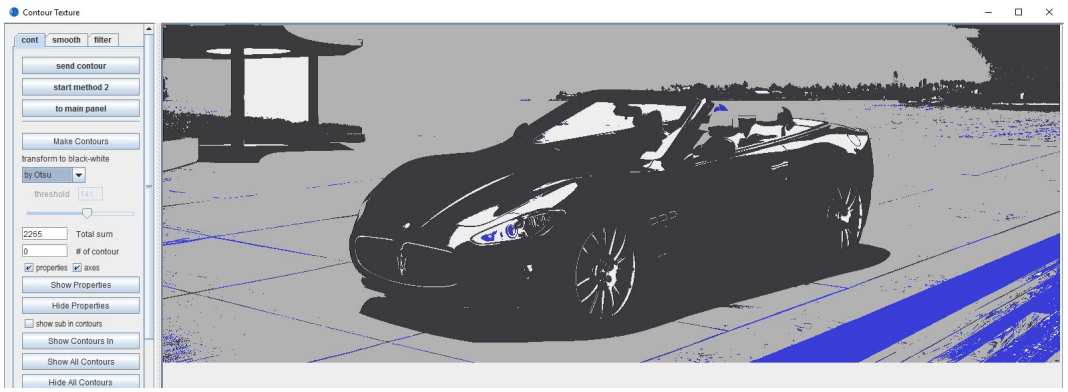

Рис. 2: Применения метода Оцу

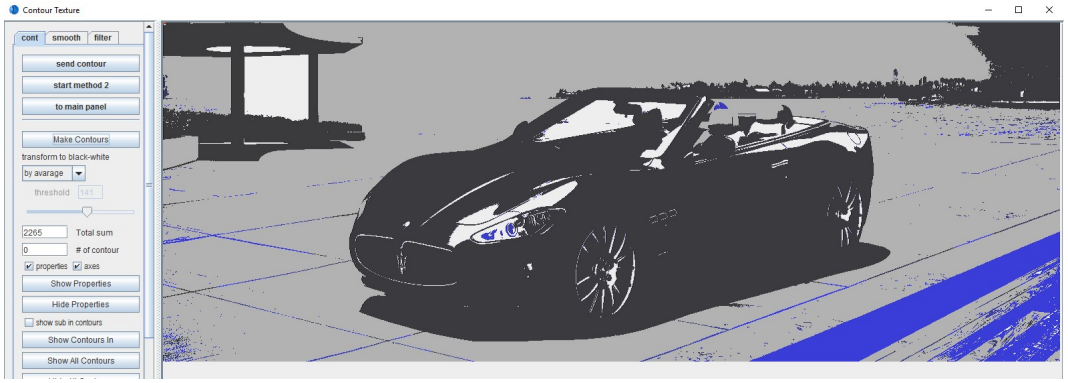

Рис. 3: Применение метода среднего порога

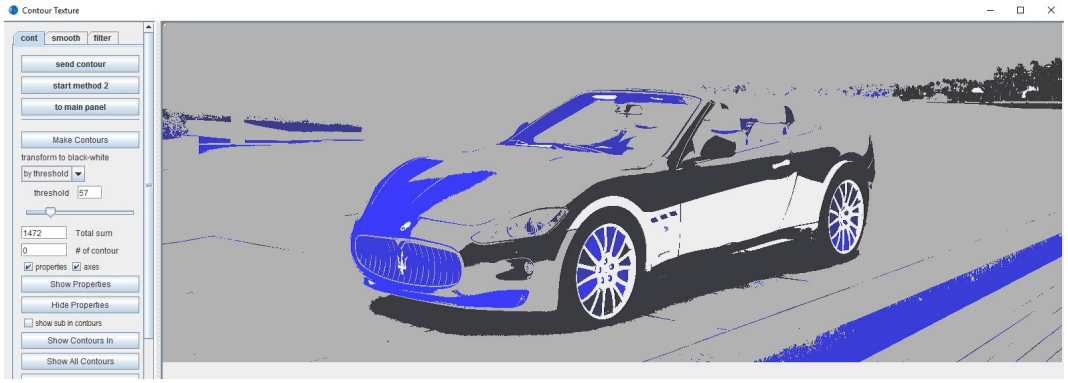

Рис. 4: Ручное изменение порога на 57

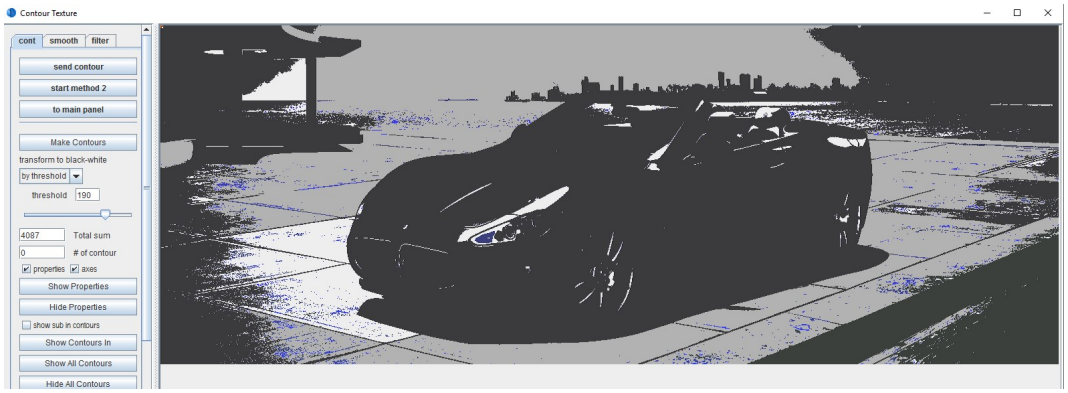

Рис. 5: Ручное изменение порога на 190

1.2. Подсчет свойств контура

Вместе с бинаризацией изображения происходит нумерация контуров и подсчет их свойств, таких как центр масс, моменты инерции, а также направление контура. Эти свойства потребуются для последующего обнаружения и отслеживания.

Пусть x_c и y_c — координаты цента масс. Обозначим за I_x и I_y следующие величины:

$$I_x = \iint\limits_S mx\,dxdy$$
$$I_y = \iint\limits_S my\,dxdy,$$

где S — площадь контура, а m — масса одного пиксела. Так как все пикселы одинаковы, то можно площадь пиксела считать за единицу площади, а массу пиксела за единицу массы. То есть можно обозначить за S количество пикселов, принадлежащих контуру. Тогда формулы I_x и I_y можно преобразовать в:

$$I_x = \sum_S x$$
$$I_y = \sum_S y,$$

Формулы x_c и y_c приобретают следующий вид:

$$x_c = \frac{I_x}{S}$$
$$y_c = \frac{I_y}{S}$$

Моменты инерции относительно начала координат (верхний левый угол изображения) вычисляются следующим образом:

$$I^0_{x^2} = \iint\limits_S x^2\,dxdy = \sum_S x^2$$
$$I^0_{y^2} = \iint\limits_S y^2\,dxdy = \sum_S y^2$$

$$I^0_{xy} = \iint\limits_S xy\,dxdy = \sum_S xy$$

Выведем формулы моментов для контура относительно его центра масс.

$$I_{x^2} = \sum_S (x - x_c)^2 = \sum_S (x^2 - 2xx_c + x_c^2) = \sum_S x^2 - 2x_c \sum_S x + Sx_c^2 =$$
$$I^0_{x^2} - 2\frac{I_x}{S} I_x + S\left(\frac{I_x}{S}\right)^2 = I^0_{x^2} - \frac{(I_x)^2}{S}$$

Аналогично для I_{y^2}:

$$I_{y^2} = I^0_{y^2} - \frac{(I_y)^2}{S}$$

Вывод формулы для смешанного момента инерции:

$$I_{xy} = \sum_S (x - x_c)(y - y_c) = \sum_S (xy - x_c y - xy_c + x_c y_c) =$$
$$\sum_S xy - x_c \sum_S y - y_c \sum_S x + Sx_c y_c = I^0_{xy} - x_c I_y - y_c I_X + Sx_c y_c = I^0_{xy} - \frac{I_x I_y}{S}$$

Так все моменты инерции представляют собой суммы, где в каждом слагаемом содержатся только координаты одного пиксела, то эти характеристики можно считатьза один проход по изображению.

Для наглядности в програме ведется подсчет направления контура. Угол поворота оси находится из перехода x и y к новым координатам:

$$x' = \cos\alpha\, x + \sin\alpha\, y$$
$$y' = -\sin\alpha\, x + \cos\alpha\, y$$

После перехода смешанный момент инерции $I_{x'y'}$ приводится к 0 и после преобразований получаем уравнение:

$$\tan 2\alpha = \frac{2I_{xy}}{I_{x^2} - I_{y^2}},$$

из которого можно получить угол поворота. На Рис. 6 представлен пример визуализации направления контура в программе с выводом всех его свойств.

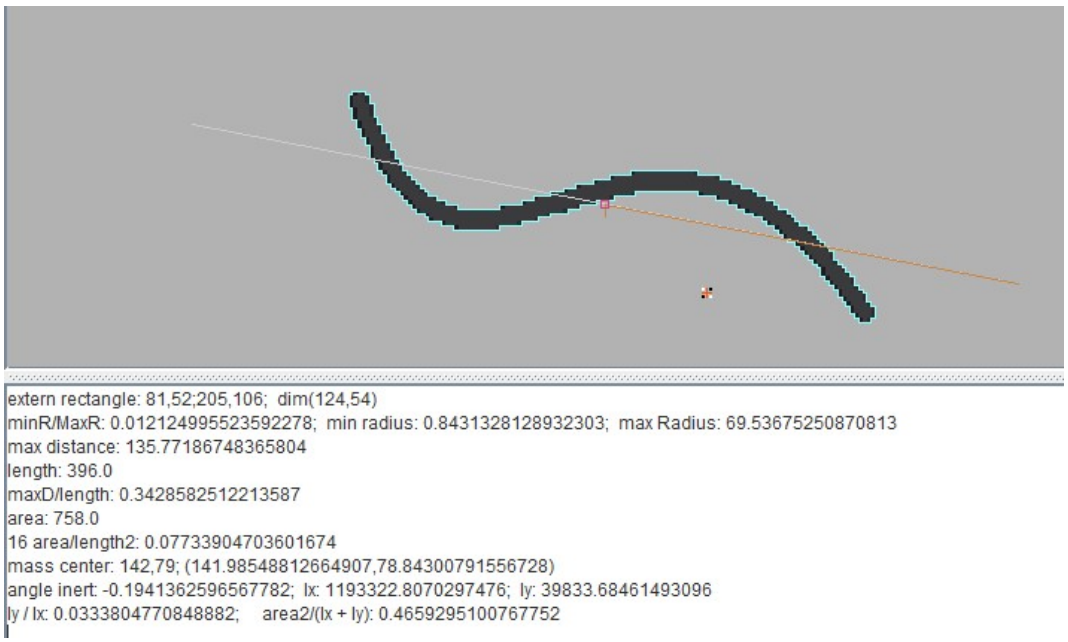

Рис. 6: Визуализация направления контура

1.3. Фильтрация контуров

После того, как одним из способов были найдены контуры, необходимо выделить те, которые максимально удовлетворяют сегментам пунктирных линий. В качестве критерия фильтрации было выбрано отношение моментов инерции контура, посчитанных в предыдущем параграфе.

Отношение моментов инерции контура для любого контура принадлежит интервалу (0, 1). Чем контур является продольнее, тем это отношение ближе к 0.

В графическом интерфейсе в интерактивном режиме с помощью бегунка можно изменять порог отношения моментов. Все объекты, отношение моментов инерции которых меньше заданого порога, будут иметь в байте прозрачности FF. Ниже на Рис. 7 представлен пример неотфильтрованного изображения, а на Рис. 8 — отфильтрованное изображение так, чтобы остались только продольные контуры.

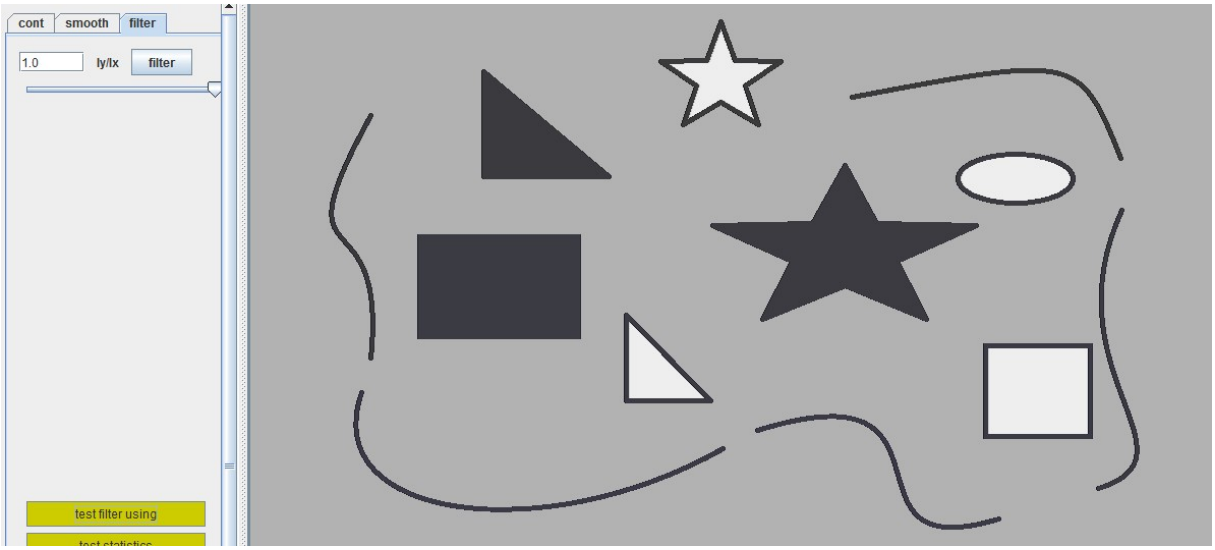

Рис. 7: Неотфильтрованые контуры с порогом 1.0

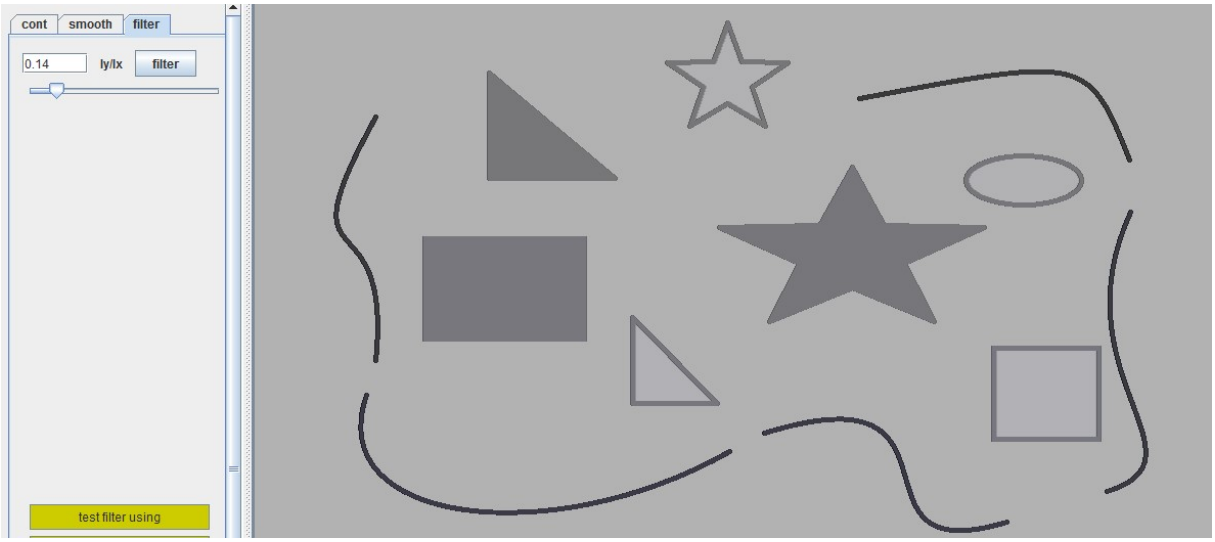

Рис. 8: Фильтрация контуров с порогом 0.14

Глава 2. Обнаружение линий и векторизация

После того, как были найдены и отфильтрованы нужные контуры, происходит их передача в окно Curve Detector Control, где реализованы остальные шаги алгоритма.

2.1. Контурная скелетизация

На вход контурной скелетизации поступает изображение, где каждый фоновый пиксель имеет значение 0x00000000, а каждый нефоновый пиксель — 0xFFNNNNNN, где NNNNNN — номер контура. Скелетизация контуров происходит через утончение фигуры с использованием алгоритма Розенфельда [6]. Идея этого алгоритма состоит в том, что последовательно перекрашиваются граничные пикселы четырех видов:

1. “северные“ граничные, имеющие фоновую точку сверху;
2. “южные“ граничные, имеющие фоновую точку снизу;
3. “западные“ граничные, имеющие фоновую точку слева;
4. “восточные“ граничные, имеющие фоновую точку справа.

Перекрашиваются в фоновый цвет только те точки, которые при удалении не нарушают связности контура. Это было реализовано следующим образом: вокруг каждого пиксела, удовлетворяющего какому-то из видов граничных пикселов на текущем шаге итерации, рассматривается окрестность из 8 пикселов. На окрестности находятся последовательности пикселов, где из каждого пиксела данной последовательности можно проложить путь по 8-ми смежности. Если таких последовательностей больше 1, то из этого следует, что при перекрашивании в фоновый цвет данного пиксела нарушится связность контура в целом. На Рис. 9 и Рис. 10 проиллюстрирован пример работы алгоритма скелетизации продольного контура. При этом у скелетизированного продольного контура у каждого пиксела не больше 2 соседей. Концы каждого отрезка обозначены кодом 0x4FNNNNNN.

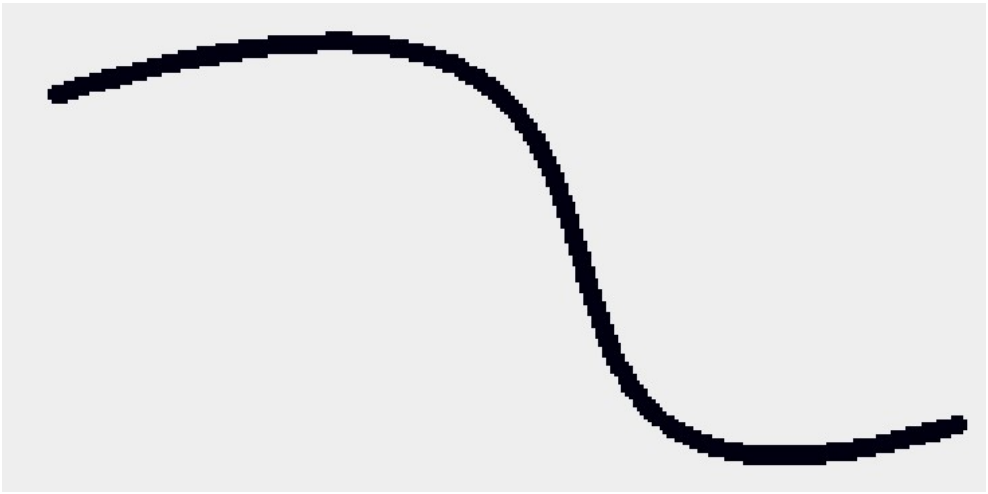

Рис. 9: Пример продольного контура

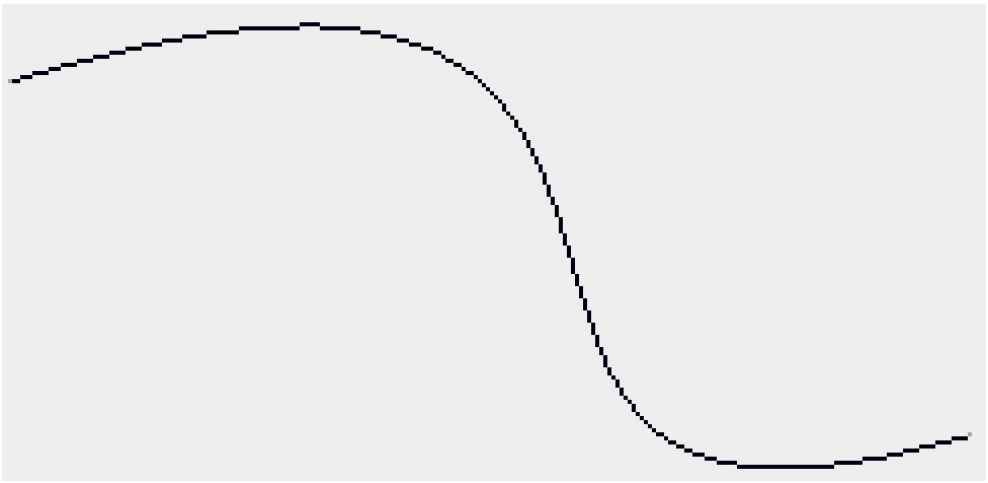

Рис. 10: Скелетизация продольного контура

В алгоритме в старший полубайт прозрачности тех пикселов, которые являются концами скелетизированных контуров (то есть имеют одного соседа), записывается код 4. Это сделано для оптимальности следующего этапа алгоритма.

На вход алгоритма может поступить объект, который состоит из пересечения двух продольных контуров, но сам по себе продольным контуром не является. При скелетизации таких контуров у некоторых пикселов может оказаться более двух соседей. Поэтому в алгоритме скелетизации идет проверка всех пикселов — если у пиксела больше двух соседей, то этот пиксел становится фоновым, а все его нефоновые соседи обозначаются кодом конца отрезка. На Рис. 11 и Рис. 12 проиллюстрирован пример работы алгоритма скелетизации пересечения продольных контуров.

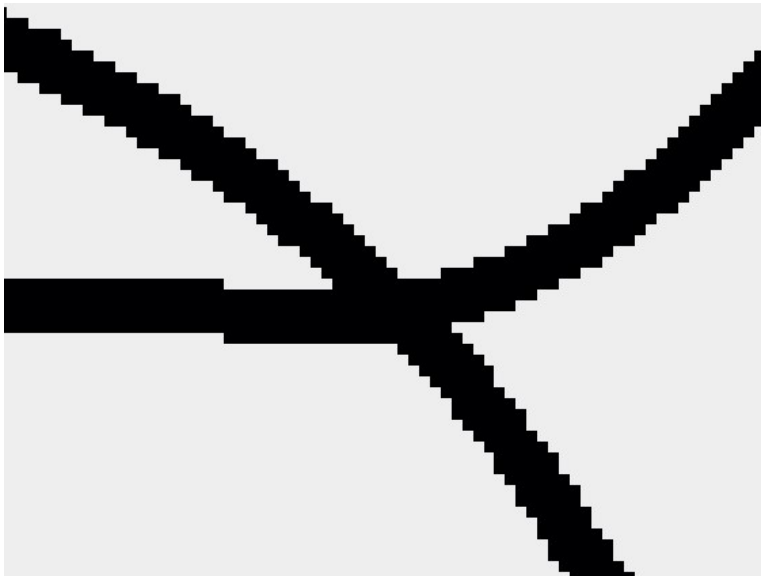

Рис. 11: Пример пересечения продольных контуров

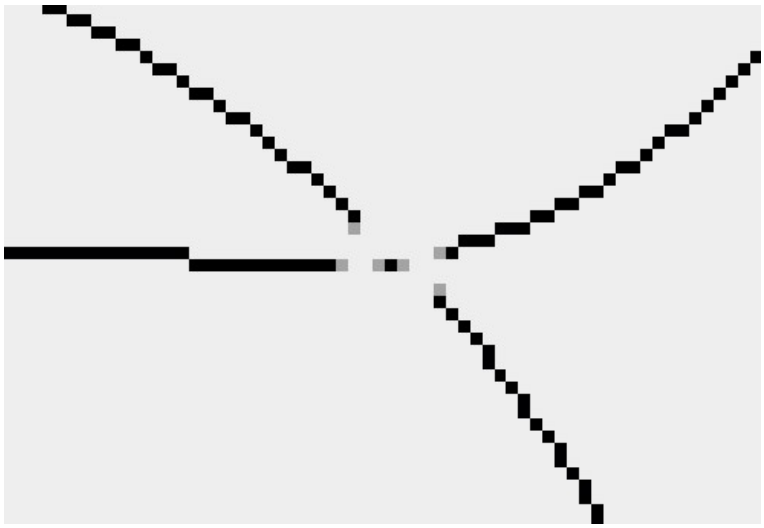

Рис. 12: Скелетизация пересечения продольных контуров

2.2. Отслеживание скелетизированных сегментов

На вход поступает изображении после скелетизации контура. Для начала создаем список уже рассмотренных концов скелетов. Во время прохода по изображению при нахождении конца отрезка начинается отслеживание до тех пор, пока не будет найден второй конец. После этого оба конца записываются в список рассмотреных и продолжается проход по изображению. На выходе алгоритм возвращает список полученных кривых. На Рис. 13 представлен пример контура после скелетизации, представленном на Рис. 12.

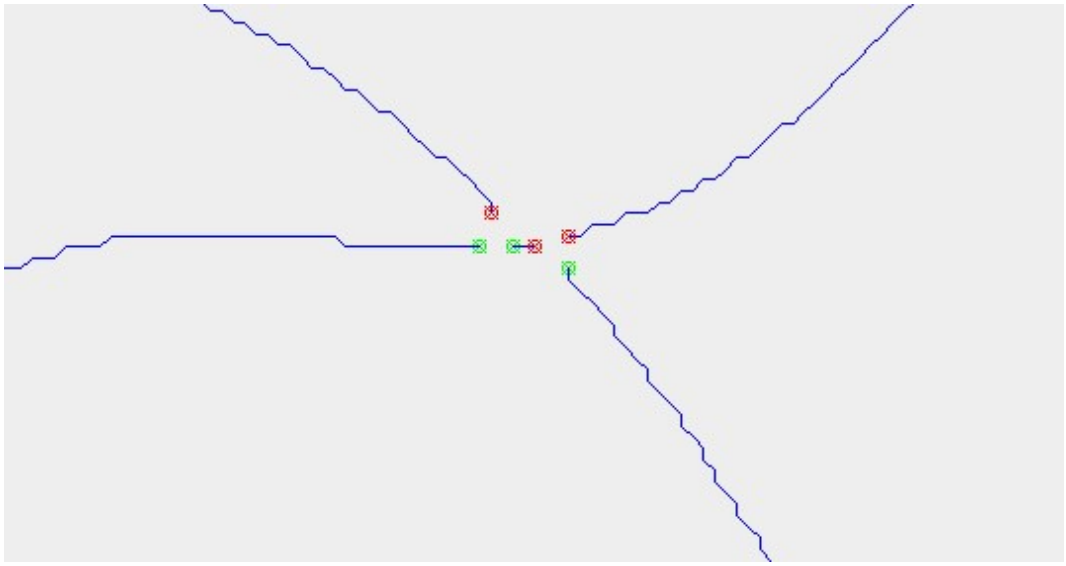

Рис. 13: Пример отслеживания

2.3. Соединение сегментов

На вход поступает список кривых и два порога — угол и расстояние. Каждая кривая содержит список точек, которые в свою очередь содержат информацию о своих координатах на изображении. Для более быстрой работы алгоритма все концы отрезков и их усредненные направления по нескольким точкам до конца кривой записываются в отдельные массивы. Затем для каждого конца ищем наиболее подходящего кандидата на соединение. На первом шаге отсеиваются все кандидаты, не удовлетворяющие порогам расстояния. На втором шаге проверяется удовлетворение направлений угловому порогу как от текущего рассматриваемого конца до конца-кандидата, так и наоборот. На третьем шаге из концов-кандидатов выбирается наиболее близкий к рассматриваему. На Рис. 14 в качестве примера приведена ситуация пересечения прерывистой линии со сплошной до и после скелетизации. На Рис. 15 отслеживание по порогам дистанции и угла хорошо отследило как сплошную, так и прерывистую линии.

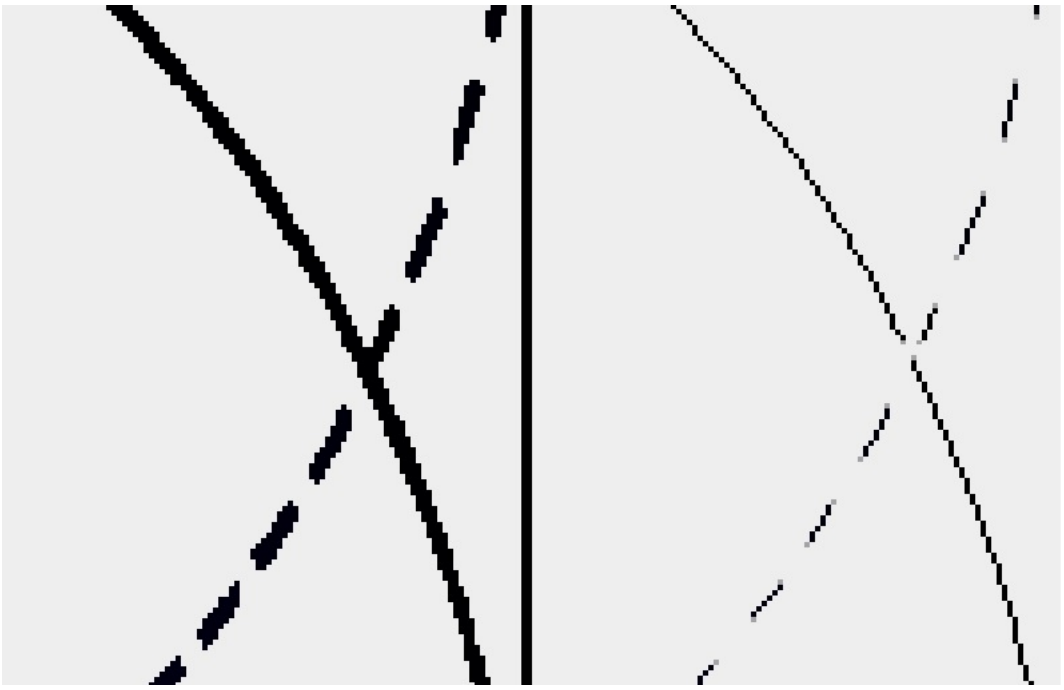

Рис. 14: Пример пересечения сплошной и прерывистой линий и его скелитизация

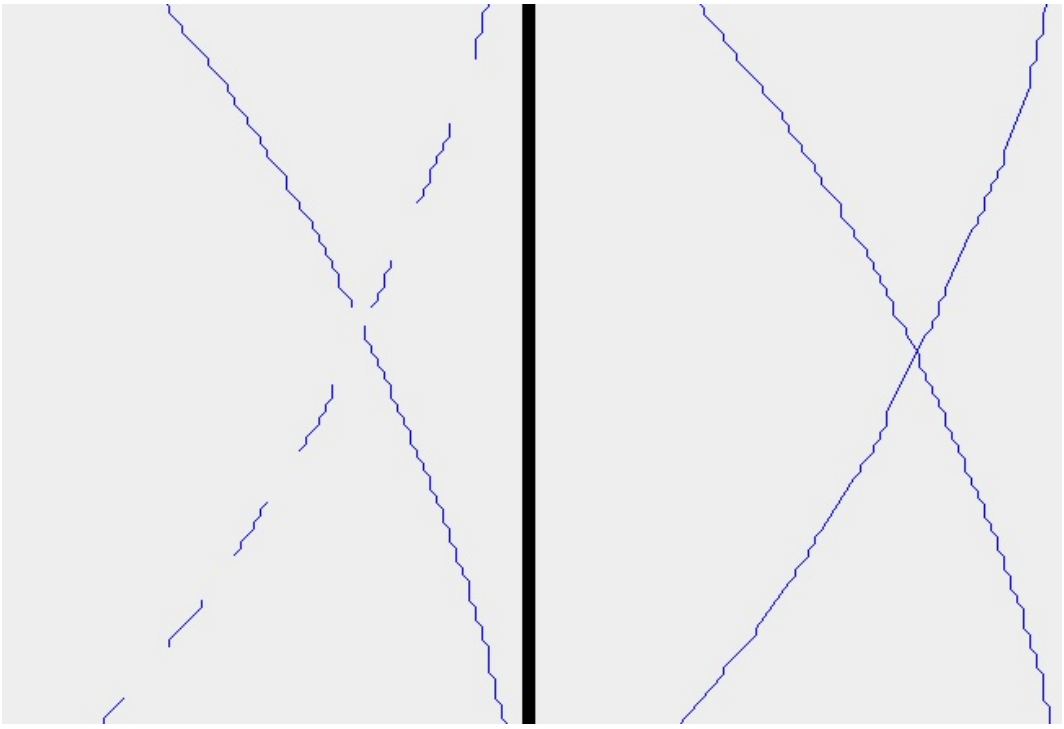

Рис. 15: Пример отслеживания и соединения сегментов с порогами дистанции 37 и угла 16° с отключенной визуализацией концов сегментов

Глава 3. Соединение линий после поиска градиента

Помимо соединения скелетизированных контуров, алгоритм соединения линий можно применить к алгоритму обнаружения и прослеживания кривых, описанному в [7]. Также в этом алгоритме сделаны некоторые изменения, о которых будет описано ниже.

3.1. Выбор оператора и модификация детектора границ

От выбора оператора зависит то, каким образом вычисляются градиенты изображения. В [7] градиенты вычислялись через оператор Кэнни [8] (только по горизонтальным и вертикальным соседям). То есть, если изображение представляет собой одномерный массив размером $width \times height$, то значения градиентов тех пикселов, которые не лежат на границе изображения, вычисляются следующим образом:

$$Grad_i^x = I_{i+1} - I_{i-1}$$
$$Grad_i^y = I_{i+width} - I_{i-width},$$

где i — индекс пиксела в массиве, $Grad$ — значение градиента в пикселе, I — яркость пиксела (после перевода изображения в полутоновое).

Для вычисления градиента пиксела лучше также использовать значения интенсивности его диагональных соседей. Было произведено сравнение нескольких операторов между собой: Кэнни [8] [9], Собеля [10] [9], Щарра [11]. В ходе экспериментов с изображениями оказалось, что вычисление градиентов с помощью оператора Щарра в совокупности с дальнейшим отслеживанием линий показало наилучший результат. Градиенты с помощью оператора Щарра вычисляются следующим образом:

$$Grad_i^x = \frac{1}{16}\Big(10(I_{i+1} - I_{i-1}) + 3(I_{i-width-1} + I_{i+width+1} - I_{i-width-1} - I_{i+width-1})\Big)$$

$$Grad_i^y = \frac{1}{16}\Bigg(10(I_{i+width} - I_{i-width}) + \\ + 3(I_{i+width-1} + I_{i-width-1} - I_{i+width+1} - I_{i-width+1})\Bigg)$$

В совокупности с выбором оператора для вычисления градиента пикселов также немаловажно правильно выбрать те пикселы, которые являются максимальным перепадом абсолютного градиента в направлении градиента. В [7] обнулялся байт прозрачности всех пикселов, кроме тех, у которых оба соседних пиксела по направлению градиента меньше оцениваемого. В модифицированном алгоритме мы пикселу значение 0, если он удовлетворяет одному из двух условий:

- Абсолютное значение градиента текущего пиксела меньше абсолютного значения градиента хотя бы одного из соседей по направлению градиента в обе стороны.
- Абсолютное значение текущего пиксела равно абсолютному значению градиента его соседа в направлении градиента.

Также дополнительно был реализован метод удаления углов, который между любыми диагональными соседями удаляет между ними смежного соседа по вертикали/горизонтали. Этот метод препятствуют появлению маленьких параллельных сегментов вдоль основной границы.

Теперь покажем на примере разницу между текущей и прошлой версиями нахождения границ. В качестве примера изображения возьмем размытый овал (Рис. 16). Именно на размытых изображениях наибольшее различие было зафиксировано при сравнении. Для большей наглядности будет продемонтрировано отслеживание границы. На Рис. 17 слева продемонтсрирован результат отслеживания границ, найденных с помощью оператора Кэнни, исходного детектора границ и без использования метода удаления углов. В сравнение этому результату Рис. 17 справа представляет собой результат отслеживания границ, найденных с помощью оператора Щарра, модифицированного детектора границ и с использованием метода удаления углов. Видно, что результат результат справа лучше, чем слева.

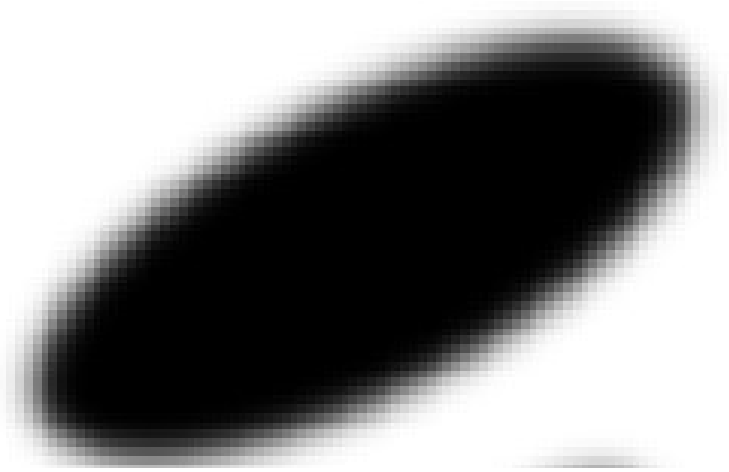

Рис. 16: Пример размытого объекта

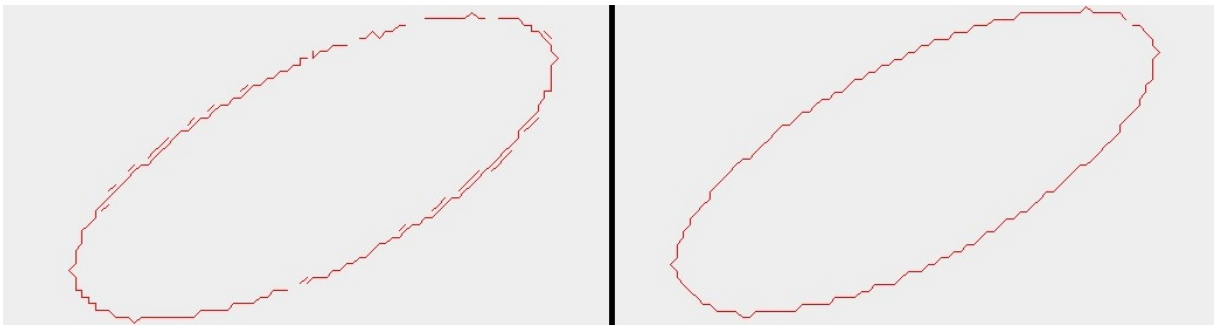

Рис. 17: Сравнение отслеживания линий после нахождения границ

3.2. Соединение линий

Так как при отслеживании границ объектов, найденных градиентным способом, могут происходить разрывы, то описанный в 2.3. алгоритм соединения сегментов можно использовать здесь, так как и в контурном отслеживании и в градиентном отслеживании формируется список экземпляров класса $Curve$, поэтому для соединения линий можно использовать один универсальный метод. При этом результаты отслеживания и соединения двух видов можно хранить и отображать вместе на одном изображении.

Чтобы различать результаты контурного и градиентного отслеживания, будем отображать их разными цветами. Красные кривые — результат градиентного отслеживания, а синие — результат контурного отслеживания. На Рис. 18 и Рис. 19 представлены исходное изображение и результат применения отслеживания и соединения кривых и контурного, и градиентного типов.

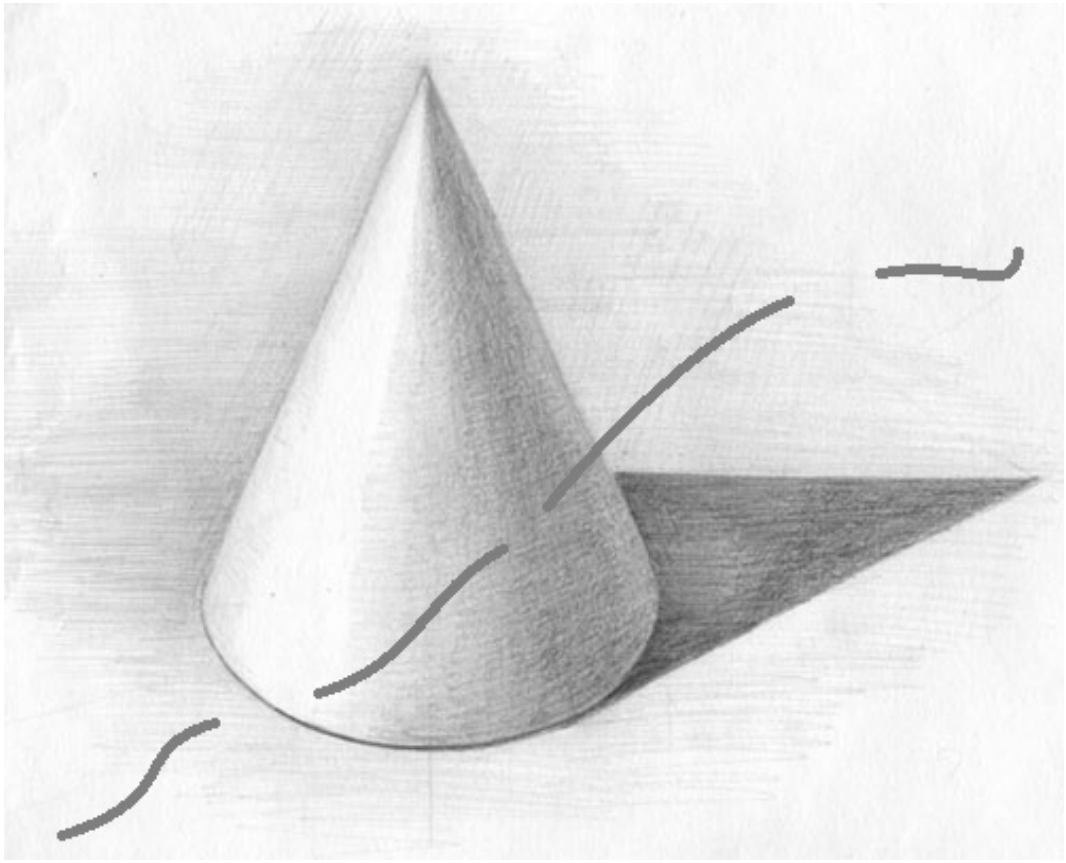

Рис. 18: Пример исходного изображения, для которого можно применить и градиентные, и контурные методы

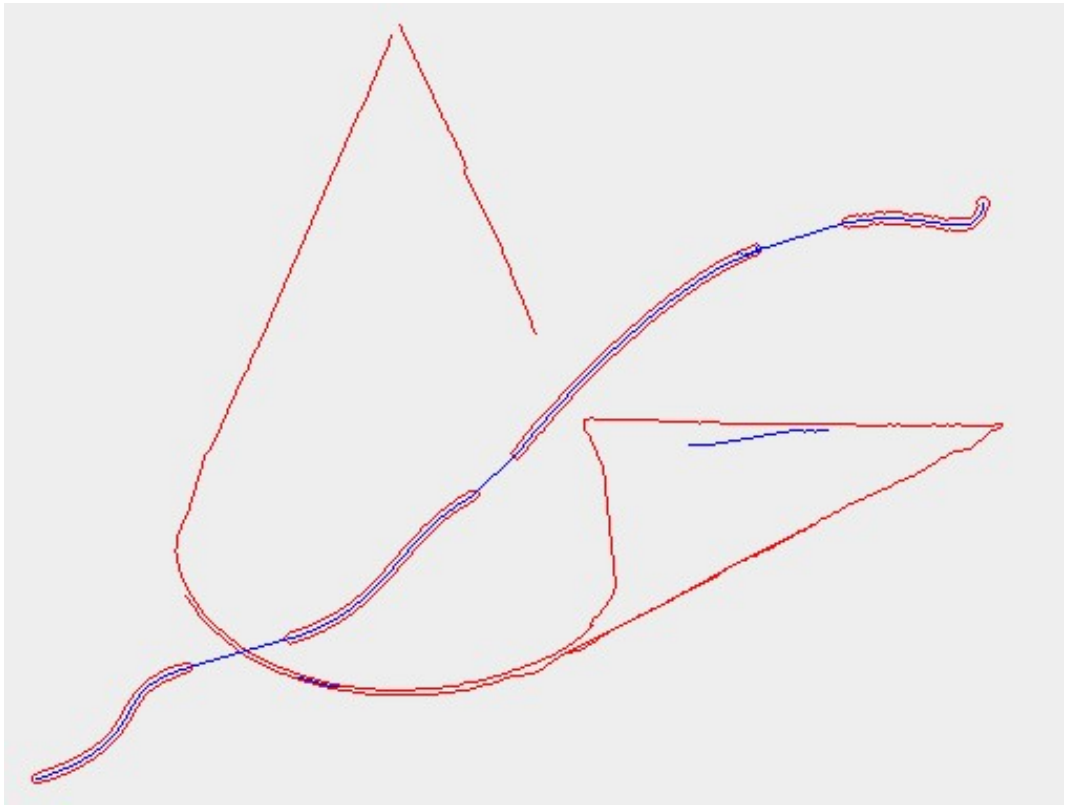

Рис. 19: Результат применения градиентных и контурных методов

Выводы

- В данное работе была разработан и реализован алгоритм обнаружения и отслеживания прерывистых линий.
- Проект включает в себя как работу с контурными методами, так и с градиентными, объединяя результаты их распознования и отслеживания на одном изображении.
- В проекте реализовано отслеживание как сплошных, так и прерывистых линий. Также разрешаются такие сложные ситуации на изображении, как пересечение кривых и пунктирных линий.
- Работа в программе происходит в графическом интерфейсе, что может быть удобно любому пользователю.
- Разработанный инструментарий может быть использован как основа для прикладных задач специалистами смежных направлений.

Заключение

Созданный инструментарий будет размещен в свободном доступе и будет доступен для скачивания. Данный проект обспечит для специалистов фундаментальную базу в области обнаружения и векторизации разрывных линий на цифровых изображениях.

К сожалению, в программе не предусамтривается автоматического подбора порогов для оптимального выделения продольных контуров и соединения сегментов, но может послужить фундаментальной базой для последующих исследований и модификаций.

Список литературы

[1] Raghupathy K. Curve tracing and curve detection in images // Cornell University, August, 2004, 124 P

[2] Lezama J., Randall G., Morel J., Grompone R. An Unsupervised Point Alignment Detection Algorithm // Image Processing On Line, 2015, P. 296–310.

[3] Lezama J., Randall G., Morel J., Grompone R. An Unsupervised Algorithm for Detecting Good Continuation in Dot Patterns // Image Processing On Line, 2016, P. 81–92.

[4] Otsu N. A Threshold Selection Method from Gray-Level Histograms // IEEE Transactions on Systems, Man, and Cybernetics, Vol. 9, No. 1, 1979, P. 62–66.

[5] Balarini J., Nesmachnow S. A C++ Implementation of Otsu's Image Segmentation Method // Image Processing On Line, 2015, P. 155–164.

[6] Местецкий Л. М. Непрерывная морфология бинарных изображений: фигуры, скелеты, циркуляры. М.: ФИЗМАТЛИТ, 2009. С. 44–46.

[7] Задорожный С. А. Обнаружение и прослеживание кривых на цифровых изображениях, 2015.

[8] Canny J. A Computational Approach to Edge Detection // IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. Pami-8, No. 6, 1986, P. 679–698.

[9] Sponton H., Cardelino J. A Review of Classic Edge Detectors // Image Processing On Line, 2015, P. 90–123.

[10] Duda R., Hart P. Pattern Classification and Scene Analysis // John Wiley and Sons, 1973, P. 271–272.

[11] Jähne B., Scharr H., Körke S. Principles of filter design // Handbook of Computer Vision and Applications 2, P. 125–151.

Приложение

А. Вычисление градиента на изображении

```
public static int[] gradient(int[] pixels, int width, int height, ↵
int gradThreshold, int bitMask) {
        double[] gray = new double[pixels.length];
        for (int k = 0; k < pixels.length; k++) {
            gray[k] = ((pixels[k] & 0x00FF0000 & bitMask) >> 16) * ↵
            0.33
                    + ((pixels[k] & 0x0000FF00 & bitMask) >> 8) * ↵
                    0.56
                    + ((pixels[k] & 0x000000FF & bitMask) >> 0) * ↵
                    0.11;
        }
        int[] grad = new int[pixels.length];
        Arrays.fill(grad, 0xFF000000);
        for (int j = 1; j < height - 1; j++) {
            for (int i = 1; i < width - 1; i++) {
                int k = j * width + i;
                // Оператор Кэнни
                //int grX = (int) (gray[k + 1] - gray[k - 1]);
                //int grY = (int) (gray[k + width] - gray[k - ↵
                width]);
                // Оператор Собеля
                //int grX = (int) (2 * (gray[k + 1] - gray[k - 1]) ↵
                + gray[k - width + 1] + gray[k + width + 1] - gray[↵
                k - width - 1] - gray[k + width - 1]) / 4;
                //int grY = (int) (2 * (gray[k + width] - gray[k - ↵
                width]) + gray[k + width - 1] - gray[k - width - 1]↵
                + gray[k + width + 1] - gray[k - width + 1]) / 4;
                // Оператор Щарра
```

```
                int grX = (int) (10 * (gray[k + 1] - gray[k - 1]) +
                3 * (gray[k - width + 1] + gray[k + width + 1] -
                gray[k - width - 1] - gray[k + width - 1])) / 16;
                int grY = (int) (10 * (gray[k + width] - gray[k -
                width]) + 3 * (gray[k + width - 1] - gray[k -
                width - 1] + gray[k + width + 1] - gray[k - width +
                1])) / 16;
                int sign = 0;
                if (grX < 0) {
                    sign |= 0x80;
                    grX = -grX;
                }
                if (grX > 0xFF) {grX = 0xFF;}
                if (grY < 0) {
                    sign |= 0x40;
                    grY = -grY;
                }
                if (grY > 0xFF) {grY = 0xFF;}

                grad[k] = 0xFF000000 | sign | (grX << 16) | (grY <<
                8);
            }
        }
        return grad;
    }
```

B. Детектор границ

```
public static int[] detectEdgesOnWeakValues(int[] grads, int width,
int height, int gradThreshold) {
    // o---------> x
```

```
    // |  8 1 2
    // |  7 X 3
    // |  6 5 4
    // V
    //   y
  for (int j = 0; j < height; j++) {
   for (int i = 0; i < width; i++) {
    int k = j * width + i;
    if (j == 0 || j == height - 1 || i == 0 || i == width - 1) {
     grads[k] &= 0x00FFFFFF;
     continue;
    }
    double tn = Math.sqrt(2) - 1;
    int gx = (grads[k] & 0xFF0000) >> 16;
    int gy = (grads[k] & 0x00FF00) >> 8;
    double abs2 = gx * gx + gy * gy;
    if (gy < gx * tn) { //3 | 7
     if (abs2 < absGrad(grads[k + 1]) || abs2 < absGrad(grads[k - ↩
     1])) {
      grads[k] &= 0x00FFFFFF;
     } else if ((grads[k] & 0xA0) == 0 && abs2 == absGrad(grads[k +↩
     1]) || (grads[k] & 0xA0) != 0 && abs2 == absGrad(grads[k - 1])↩
     ) {
      grads[k] &= 0x00FFFFFF;
     }
    } else if (gx < gy * tn) { //1 | 5
     if (abs2 < absGrad(grads[k + width]) || abs2 < absGrad(grads[↩
     k - width])) {
      grads[k] &= 0x00FFFFFF;
     } else if ((grads[k] & 0x50) == 0 && abs2 == absGrad(grads[k +↩
     width]) || (grads[k] & 0x50) != 0 && abs2 == absGrad(grads[k -↩
```

```
    width])) {
     grads[k] &= 0x00FFFFFF;
    }
   } else if ((grads[k] & 0xA0) == 0 && (grads[k] & 0x50) == 0 || ↩
   (grads[k] & 0xA0) != 0 && (grads[k] & 0x50) != 0) { //4 | 8
    if (abs2 < CurveDetectMethods.absGrad(grads[k - width - 1]) ↩
    || abs2 < CurveDetectMethods.absGrad(grads[k + width + 1])) {
     grads[k] &= 0x00FFFFFF;
    } else if ((grads[k] & 0xA0) == 0 && (grads[k] & 0x50) == 0 ↩
    && abs2 == absGrad(grads[k + width + 1]) || (grads[k] & 0xA0) ↩
    != 0 && (grads[k] & 0x50) != 0 && abs2 == absGrad(grads[k - ↩
    width - 1])) {
     grads[k] &= 0x00FFFFFF;
    }
   } else if ((grads[k] & 0xA0) == 0 && (grads[k] & 0x50) != 0 || ↩
   (grads[k] & 0xA0) != 0 && (grads[k] & 0x50) == 0) { // 2 | 6
    if (abs2 < CurveDetectMethods.absGrad(grads[k - width + 1]) ↩
    || abs2 < CurveDetectMethods.absGrad(grads[k + width - 1])) {
     grads[k] &= 0x00FFFFFF;
    } else if ((grads[k] & 0xA0) == 0 && (grads[k] & 0x50) != 0 ↩
    && abs2 == absGrad(grads[k - width + 1]) || (grads[k] & 0xA0) ↩
    != 0 && (grads[k] & 0x50) == 0 && abs2 == absGrad(grads[k + ↩
    width - 1])) {
     grads[k] &= 0x00FFFFFF;
    }
   }
  }
 }
 for (int i = 0; i < grads.length; ++i) {
  if (grads[i] >>> 24 == 0) {
   grads[i] = 0;
```

```
  }
 }
 return grads;
}
```

C. Алгоритм скелетизации контуров

```
public static int[] erosion(int[] pixels, int width) {
 final int BACKGROUND = 0;
 final int heigth = pixels.length / width;
 // зафонивание рамки вокруг ↵
 изображения (во избежание ↵
 конфликтов на границе)
 for (int i = 0; i < pixels.length; ++i) {
  if (i < width || i >= pixels.length - width || i % width == 0 ↵
  || i % width == width - 1) {
   pixels[i] = BACKGROUND;
  }
 }
 // direction - с какой стороны мы удаляем ↵
 пикселы.
 // 0 - север
 // 1 - юг
 // 2 - запад
 // 3 - восток
 boolean isErosionComplete = false;
 while (!isErosionComplete) {
  isErosionComplete = true;
  for (int direction = 0; direction < 4; direction++) {
   // список индексов которые ↵
   подлежать закрашиванию BACKGROUND
```

```
List<Integer> listOfIndexes = new LinkedList<>();
// индекс направления соседа ⤶
пиксела относительно ⤶
центрального
int indexOfNeighbourBackground = -1;
switch (direction) {
 case 0:
  indexOfNeighbourBackground = -width;
  break;
 case 1:
  indexOfNeighbourBackground = width;
  break;
 case 2:
  indexOfNeighbourBackground = -1;
  break;
 case 3:
  indexOfNeighbourBackground = 1;
  break;
 default:
  throw new IndexOutOfBoundsException(⤶
  "indexOfNeighbourBackground = " + indexOfNeighbourBackground)⤶
  ;
}
for (int s = 1; s < heigth - 1; ++s) {
 for (int l = 1; l < width - 1; ++l) {
  int i = s * width + l;
  if (pixels[i] != BACKGROUND && pixels[i + ⤶
  indexOfNeighbourBackground] == BACKGROUND) {
   // проверяем, не нарушит ли ⤶
   пиксель связности при удалении
   int numberOfNeighboursBackground = 0;
```

```
for (int n = i - width; n <= i + width; n += width) {
 for (int m = -1; m <= 1; m++) {
  if (pixels[n + m] == BACKGROUND) {
   numberOfNeighboursBackground++;
  }
 }
}
if (numberOfNeighboursBackground == 7) {
 pixels[i] &= 0x4FFFFFFF;
 pixels[i] |= 0x40000000;
 continue;
}
// обходим пикселы по кругу
// 0 1 2
// 7 - 3
// 6 5 4
// считаем количество "кластеров",
т.е. тех групп не фоновых
// пикселов, которые окажутся не
связными (по 8-ми смежности)
// при удалении центрального
int indexOfIteratingPixel = 0;
int realIndexOfIteratingPixel = 0;
int realIndexOfNextPixel = 0;
int realIndexOfAfterNextPixel = 0;
int amountOfClusters = 0;
boolean isIncrementAmountOfClusters = true;
while (indexOfIteratingPixel < 8) {
 switch (indexOfIteratingPixel) {
  case 0:
   realIndexOfIteratingPixel = i - width - 1;
```

```
 realIndexOfNextPixel = i - width;
 realIndexOfAfterNextPixel = i - width + 1;
 break;
case 1:
 realIndexOfIteratingPixel = i - width;
 realIndexOfNextPixel = i - width + 1;
 realIndexOfAfterNextPixel = i + 1;
 break;
case 2:
 realIndexOfIteratingPixel = i - width + 1;
 realIndexOfNextPixel = i + 1;
 realIndexOfAfterNextPixel = i + width + 1;
 break;
case 3:
 realIndexOfIteratingPixel = i + 1;
 realIndexOfNextPixel = i + width + 1;
 realIndexOfAfterNextPixel = i + width;
 break;
case 4:
 realIndexOfIteratingPixel = i + width + 1;
 realIndexOfNextPixel = i + width;
 realIndexOfAfterNextPixel = i + width - 1;
 break;
case 5:
 realIndexOfIteratingPixel = i + width;
 realIndexOfNextPixel = i + width - 1;
 realIndexOfAfterNextPixel = i - 1;
 break;
case 6:
 realIndexOfIteratingPixel = i + width - 1;
 realIndexOfNextPixel = i - 1;
```

```
    realIndexOfAfterNextPixel = i - width - 1;
    break;
   case 7:
    realIndexOfIteratingPixel = i - 1;
    realIndexOfNextPixel = i - width - 1;
    realIndexOfAfterNextPixel = i - width;
    break;
   default:
    throw new UnsupportedOperationException("Error of ↩
    indexOfInterator: " + indexOfIteratingPixel);
  }
  if (pixels[realIndexOfIteratingPixel] != BACKGROUND) {
   if (isIncrementAmountOfClusters) {
    amountOfClusters++;
    isIncrementAmountOfClusters = false;
   }
   if (indexOfIteratingPixel % 2 != 0 && pixels[↩
   realIndexOfAfterNextPixel] != BACKGROUND) {
    indexOfIteratingPixel += 2;
    if (indexOfIteratingPixel > 7)
     amountOfClusters--;
   } else {
    indexOfIteratingPixel++;
    if (indexOfIteratingPixel > 7 && pixels[↩
    realIndexOfNextPixel] != BACKGROUND)
     amountOfClusters--;
   }
  } else {
   isIncrementAmountOfClusters = true;
   indexOfIteratingPixel++;
  }
```

```
      }
      // если количесво "кластеров" ⤶
      больше 2, то при удалении ⤶
      нарушается
      // связность => оставляем
      if (amountOfClusters < 2) {
       isErosionComplete = false;
       listOfIndexes.add(i);
      }
     }
    }
   }
   // делаем фоном
   listOfIndexes.stream().forEach((integer) -> {
    pixels[integer] = BACKGROUND;
   });
 }
}
// список пикселов на удаление
List<Integer> list = new LinkedList<>();
// так как конечная цель ⤶
отслеживание кривых, то если у ⤶
пиксела
// больше 2 соседей то его удаляем, а ⤶
соседние пикселы помечаем как ⤶
концы
// (у соседей должно остаться по ⤶
одному соседу)
for (int i = 1; i < heigth - 1; ++i) {
 for (int j = 1; j < width - 1; ++j) {
  int currentIndex = i * width + j;
```

```
  if (pixels[currentIndex] == BACKGROUND)
   continue;
  byte numberOfNeighbours = 0;
  for (int s = -1; s <= 1; ++s) {
   for (int l = -1; l <= 1; ++l) {
    if (s == 0 && l == 0) continue;
    if (pixels[currentIndex + s * width + l] != BACKGROUND)
     numberOfNeighbours++;
   }
  }
  if (numberOfNeighbours > 2) {
   list.add(currentIndex);
  }
 }
}
// делаем фоном
list.stream().forEach((integer) -> {
 pixels[integer] = BACKGROUND;
});
for (int i = 1; i < heigth - 1; ++i) {
 for (int j = 1; j < width - 1; ++j) {
  int currentIndex = i * width + j;
  if (pixels[currentIndex] == BACKGROUND)
   continue;
  byte numberOfNeighbours = 0;
  for (int s = -1; s <= 1; ++s) {
   for (int l = -1; l <= 1; ++l) {
    if (s == 0 && l == 0) continue;
    if (pixels[currentIndex + s * width + l] != BACKGROUND)
     numberOfNeighbours++;
   }
```

```
    }
    if (numberOfNeighbours == 1) {
     pixels[currentIndex] &= 0x4FFFFFFF;
     pixels[currentIndex] |= 0x40000000;
    }
   }
  }
  return pixels;
 }
```

D. Отслеживание скелетезированных контуров

```
 public static ArrayList<Curve> contourTracking(int[] pixels, int ↩
width, int height) {
  ArrayList<Curve> contourCurves = new ArrayList<>();
  // временно храним концы скелетов (↩
  уже обработанные)
  // код конца отрезка - 0x4NNNNNNN
  List<Integer> ends = new LinkedList<>();
  for (int j = 1; j < height - 1; j++) {
   for (int i = 1; i < width - 1; i++) {
    //Номер текущей точки в массиве grads.
    int k = j * width + i;
    if ((pixels[k] >>> 28) == 0x4 && !ends.contains(k)) {
     Curve c = new Curve();
     c.add(new Point2D.Double(i, j));
     int currentIndex = k;
     int previousIndex = -1;
     do {
      boolean isContinue = true;
      for (int n = -1; n <= 1 && isContinue; ++n) {
```

```
        for (int m = -1; m <= 1 && isContinue; ++m) {
         if (n == m && n == 0) continue;
         int tempIndex = currentIndex + n * width + m;
         if (pixels[tempIndex] != 0 && tempIndex != previousIndex) {
          c.add(new Point2D.Double(currentIndex % width + m, ↩
          currentIndex / width + n));
          previousIndex = currentIndex;
          currentIndex = tempIndex;
          isContinue = false;
         }
        }
       } while (pixels[currentIndex] >>> 28 != 0x4);
       if (c.getPoints().size() > 1) { // TODO фильтр по ↩
       длине curve
        contourCurves.add(c);
        ends.add(k);
        ends.add(currentIndex);
       }
      }
     }
    }
   return contourCurves;
}
```

E. Соединение отслеженных сегментов

```
 public static ArrayList<Curve> nonSolidContourTracking(ArrayList<↩
Curve> inputCurves, double distanceThreshold, double ↩
angleThreshold, int width, int height) {
  ArrayList<Curve> curves = (ArrayList<Curve>) inputCurves.clone();
```

```
if (distanceThreshold == 0 || angleThreshold == 0)
 return curves;
// сколько точек от конца отрезка мы ⤶
берем для подсчета усредненного ⤶
направления TODO
final int maxAmountOfPointsForEndDirection = 5;
// создаем массив с концами всех ⤶
кривых
// index / 2 в ends = index в curves
Point2D.Double[] ends = new Point2D.Double[curves.size() * 2];
for (int i = 0; i < ends.length; i += 2) {
 ArrayList<Point2D.Double> tempList = curves.get(i / 2).⤶
 getPoints();
 ends[i] = tempList.get(0);
 ends[i + 1] = tempList.get(tempList.size() - 1);
}
// подсчитаем усредненные ⤶
направления на всех концах (от 0 до ⤶
2pi)
// при соединении оба конца должны ⤶
удовлетворять angleThreshold
double[] directionsOfEnds = new double[ends.length];
for (int i = 0; i < directionsOfEnds.length; i += 2) {
 ArrayList<Point2D.Double> curvePoints = curves.get(i / 2).⤶
 getPoints();
 int curveSize = curvePoints.size();
 int amountOfPointsForEndDirection = curveSize < ⤶
 maxAmountOfPointsForEndDirection ?
   curveSize : maxAmountOfPointsForEndDirection;
 if (curveSize < amountOfPointsForEndDirection + 1) {
  ends[i] = null;
```

```
 ends[i + 1] = null;
 continue;
}
double sumAngle = 0;
// конец из начала списка
for (int j = amountOfPointsForEndDirection; j >= 0; --j) {
 double dx = curvePoints.get(j).x - curvePoints.get(j + 1).x;
 double dy = curvePoints.get(j + 1).y - curvePoints.get(j).y;
 //инвертируем
 sumAngle += Math.atan(dy / dx);
 if (dx < 0) {
  sumAngle += Math.PI;
 } else if (dy < 0) {
  sumAngle += 2 * Math.PI;
 }
}
directionsOfEnds[i] = sumAngle / (double)
amountOfPointsForEndDirection;
sumAngle = 0;
// второй конец
for (int j = curveSize - amountOfPointsForEndDirection; j <
curveSize; ++j) {
 double dx = curvePoints.get(j).x - curvePoints.get(j - 1).x;
 double dy = curvePoints.get(j - 1).y - curvePoints.get(j).y;
 //инвертируем
 sumAngle += Math.atan(dy / dx);
 if (dx < 0) {
  sumAngle += Math.PI;
 } else if (dy < 0) {
  sumAngle += 2 * Math.PI;
 }
```

```
 }
 directionsOfEnds[i + 1] = sumAngle / (double) ↩
 amountOfPointsForEndDirection;
}
for (int i = 0; i < ends.length; ++i) {
 if (ends[i] == null) continue;
 // флаг на то, что один из углов ↩
 выйдет за [0; 2pi]
 boolean angleFlag1 = false;
 Curve currentCurve = curves.get(i / 2);
 boolean isCurrentPointFirst = i % 2 == 0;
 Point2D.Double currentPoint = isCurrentPointFirst ?
   currentCurve.getPoints().get(0) :
   currentCurve.getPoints().get(currentCurve.getPoints().size() -↩
   1);
 // две ограничивающих линий по ↩
 angleThreshold (от 0 до 2pi)
 double angle11 = directionsOfEnds[i] - angleThreshold;
 if (angle11 < 0) {
  angle11 += 2 * Math.PI;
  angleFlag1 = true;
 }
 double angle12 = directionsOfEnds[i] + angleThreshold;
 if (angle12 > 2 * Math.PI) {
  angle12 -= 2 * Math.PI;
  angleFlag1 = true;
 }
 // будем искать минимальный угол
 double minDistance = Double.MAX_VALUE;
 Point2D.Double connectPoint = null;
 int indexInEnds = -1;
```

```
for (int j = 0; j < ends.length; ++j) {
 if (j == i || ends[j] == null) continue;
 // рассматриваем другие точки
 Point2D.Double potentialPoint = j % 2 == 0 ?
  curves.get(j / 2).getPoints().get(0) :
  curves.get(j / 2).getPoints().get(curves.get(j / 2).getPoints(
  ).size() - 1);
 boolean angleFlag2 = false;
 // считаем расстояние
 double distance = Math.sqrt(Math.pow(potentialPoint.x -
 currentPoint.x, 2) +
   Math.pow(potentialPoint.y - currentPoint.y, 2));
 if (distance > distanceThreshold) continue;
 // считаем угол от первой линии ко
 второй
 double dx = potentialPoint.x - currentPoint.x;
 double dy = currentPoint.y - potentialPoint.y;
 //инвертируем
 double angleOfPotentialPoint12 = Math.atan(dy / dx);
 if (dx < 0) {
  angleOfPotentialPoint12 += Math.PI;
 } else if (dy < 0) {
  angleOfPotentialPoint12 += 2 * Math.PI;
 }
 // если не удовлетворяет angleThreshold
 if ((!angleFlag1 || (angleOfPotentialPoint12 <= angle11 &&
 angleOfPotentialPoint12 >= angle12)) && (
 angleOfPotentialPoint12 <= angle11 || angleOfPotentialPoint12 >
 = angle12))
  continue;
 // две ограничивающих линий по
```

```
angleThreshold (от 0 до 2pi)
double angle21 = directionsOfEnds[j] - angleThreshold;
if (angle21 < 0) {
 angle21 += 2 * Math.PI;
 angleFlag2 = true;
}
double angle22 = directionsOfEnds[j] + angleThreshold;
if (angle22 > 2 * Math.PI) {
 angle22 -= 2 * Math.PI;
 angleFlag2 = true;
}
// угол от второй к первой (от 0 до 2pi)
double angleOfPotentialPoint21;
if (angleOfPotentialPoint12 < Math.PI) {
 angleOfPotentialPoint21 = angleOfPotentialPoint12 + Math.PI;
} else {
 angleOfPotentialPoint21 = angleOfPotentialPoint12 - Math.PI;
}
// если не удовлетворяет условию ⤸
от второй линии к первой
if (!((angleFlag2 && (angleOfPotentialPoint21 > angle21 || ⤸
angleOfPotentialPoint21 < angle21)) || (⤸
angleOfPotentialPoint21 > angle21 && angleOfPotentialPoint21 < ⤸
angle22)))
 continue;
// если удовлетворяет порогу угла ⤸
от первой линии ко второй
if ((angleFlag1 && (angleOfPotentialPoint12 > angle11 || ⤸
angleOfPotentialPoint12 < angle12)) || (⤸
angleOfPotentialPoint12 > angle11 && angleOfPotentialPoint12 < ⤸
angle12)) {
```

```
      if (distance < minDistance) {
       minDistance = distance;
       connectPoint = potentialPoint;
       indexInEnds = j;
      }
     }
    }
    if (connectPoint == null) continue;
    Curve testCurve = new Curve();
    testCurve.add(currentPoint);
    testCurve.add(connectPoint);
    curves.add(testCurve);
    ends[i] = null;
    ends[indexInEnds] = null;
   }
   return curves;
  }
```